On Finding Short Cycles in Cryptographic Algorithms

Elena Dubrova[1] and Maxim Teslenko[2]

[1] Royal Institute of Technology, Electrum 229, 164 40 Stockholm, Sweden
dubrova@kth.se
[2] Ericsson Research, Ericsson, Färögatan 6, 164 80 Stockholm, Sweden
maxim.teslenko@ericsson.com

Abstract. We show how short cycles in the state space of a cryptographic algorithm can be used to mount a fault attack on its implementation which results in a full secret key recovery. The attack is based on the assumption that an attacker can inject a transient fault at a precise location and time of his/her choice and more than once. We present an algorithm which uses a SAT-based bounded model checking for finding all short cycles of a given length. The existing Boolean Decision Diagram (BDD) based algorithms for finding cycles have limited capacity due to the excessive memory requirements of BDDs. The simulation-based algorithms can be applied to larger problem instances, however, they cannot guarantee the detection of all cycles of a given length. The same holds for general-purpose SAT-based model checkers. The presented algorithm can find all short cycles in cryptographic algorithms with very large state spaces. We evaluate it by analyzing Trivium, Bivium, Grain-80 and Grain-128 stream ciphers. The analysis shows these ciphers have short cycles whose existence, to our best knowledge, was previously unknown.

Keywords: Shift register, stream cipher, Trivium, Grain, cycle, SAT, fault attack, fault injection.

1 Introduction

SAT solvers are a powerful tool for finding solutions to Boolean satisfiability problem. Using backtracking and different heuristics, these solvers explore the exponential space of variable assignments looking for a satisfying assignment. In spite of NP-completeness of the SAT problem [1], today's SAT solvers are able to handle problem instances involving thousands of variables. This is sufficient for many practical SAT problems in formal verification, automatic test pattern generation, and logic synthesis [2].

In the past there were multiple attempts to use SAT solvers in cryptanalysis, including finding preimages for hash functions [3–5], generating collisions for hash functions [6], faking RSA signatures [7], recovering RSA private keys [8], finding (or disproving the existence of) weak keys in stream and block ciphers [9, 10], guess-and-determine attacks on stream ciphers [9], algebraic attacks [11–13],

and side-channel attacks on software implementations of block ciphers [14]. So called *logical cryptanalysis* has been introduced by Massacci and Marraro [15] as a general framework for encoding the low-level properties of cryptographic algorithms as SAT problems and then using efficient automated theorem-proving systems and SAT-solvers for reasoning about them. This approach has been followed by many, including [16–19]. However, it is generally quite difficult to make SAT solvers to produce meaningful results except for strongly reduced instances of cryptographic algorithms.

In this paper, we present a SAT-based algorithm which is able to produce results for important stream ciphers such as Trivium [20] and Grain [21] which are among the candidates that have continuously attracted attention since the end of the eStream project [22]. Given that the internal state space of Trivium is of size 2^{288}, designing an algorithm which guarantees that *all* cycles of a given length are detected is far from trivial. To the best of our knowledge, none of the existing algorithms is able to solve this problem. The existing Boolean Decision Diagram (BDD) based approaches [23, 24] have limited capacity due to the excessive memory requirements of BDDs. The simulation-based algorithms [25–28] can be applied to larger problem instances, however, they cannot guarantee the detection of all cycles of a given length, even if the length is short. The same holds for the SAT-based algorithms that do not limit cycle length [29] and general-purpose SAT-based model checkers, e.g. [30]. There are many algorithms for finding cycles of length one, including [31–33], but they cannot handle larger cycle lengths.

The presented algorithm finds cycles using a SAT-based bounded model checking approach [30]. A SAT-solver is used for the identification of paths of a particular fixed length k in the state transition graph of a deterministic finite state machine. The input fed into the SAT-solver is a propositional formula representing the unfolding of the transition relation by k time steps. A satisfying assignment to this propositional formula corresponds to a valid path of length k in the state transition graph. If the last state of this path is equal to its first state, we can conclude that we found a cycle of length k or a factor of k. We can then mark all cycle's states by adding them as constrains to the input formula of the SAT-solver and, in the following iterations, search only for paths in which the last state is not marked. We repeat the process iteratively until the SAT-solver cannot find any more satisfying assignments.

We evaluated the presented algorithm by analyzing Trivium, Bivium, Grain family stream ciphers and found that they have short cycles whose existence, to our best knowledge, was previously unknown (except for all-0 cycles). We describe how short cycles can be used to mount a fault attack which results in a full secret key recovery. This attack is based on the assumption that an attacker can inject a transient fault at a precise location and time of his/her choice and more than once. The fact that such an assumption can be realistic for single faults has been demonstrated by Skorobogatov and Anderson already in 2002 [34]. They used a focused flash light to set flip-flops implementing individual SRAM cells in a microcontroller to a fixed value. Clearly in 15 years

feature sizes of transistors scaled considerably, making an individual flip-flop a more difficult target. However, the equipment and methods for performing fault attacks advanced as well (see [35–37] for recent overviews). A successful laser fault injection attack capable of inducing single faults in a successive manner at the same location has been reported in 2010 for the CRT-RSA running on a 32-bit ARM Cortex-M3 core [38]. A fault injection attack using two lasers to inject two faults simultaneously at different locations has been reported in 2016 for the AES implemented in a Xilinx Spartan-6 FPGA manufactured in 45 nm technology [39]. Therefore, it seems very likely that the attack we present might be feasible in practice provided that a chip is not protected against fault injection. Therefore, it is important to bring this problem to the attention of research community.

The paper is organized as follows. Section 2 gives the background on shift registers, stream ciphers and SAT-solvers. Section 3 presents the new SAT-based algorithm. Section 4 summarizes the experimental results for four stream ciphers. In Section 5 we describe how short cycles can be used to recover a secret key by a fault attack. Section 6 concludes the paper and discusses open problems.

2 Background

This section describes basic notions used in the paper and gives a background on stream ciphers and SAT solvers. All stream we consider in this paper are based on shift registers, therefore we start by introducing shift resisters.

2.1 Shift registers

As a model of a shift register, we use a deterministic finite state machine with a set of states $S = \{0,1\}^n$ and the state transition function $f : S \to S$. We are not concerned with inputs and outputs in this paper, so we assume that the input alphabet is empty and there is no output function.

The state transition function f defines the next state $s^+ = (x_1^+, x_2^+, \ldots, x_n^+)$ as $f(s)$, where $s = (x_1, x_2, \ldots, x_n)$ is a current state and $x_i \in \{0,1\}$ and $x_i^+ \in \{0,1\}$ are Boolean variables representing the bit number i of the current and next state, respectively, $\forall i \in \{1,2,\ldots,n\}$. The bit number i of the output of f is computed by the Boolean function $f_i : S \to \{0,1\}$, $\forall i \in \{1,2,\ldots,n\}$. In other words, the state transition function f defines the mapping

$$\begin{pmatrix} x_1 \\ \ldots \\ x_n \end{pmatrix} \to \begin{pmatrix} f_1(x_1,\ldots,x_n) \\ \ldots \\ f_n(x_1,\ldots,x_n) \end{pmatrix}. \tag{1}$$

Note that such a general definition of shift registers includes the linear and non-linear feedback shift registers in the Fibonacci and the Galois configurations [40], shift registers with feedforward connections (e.g. used in Trivium), as well as Golomb's binary machines [41, p. 21].

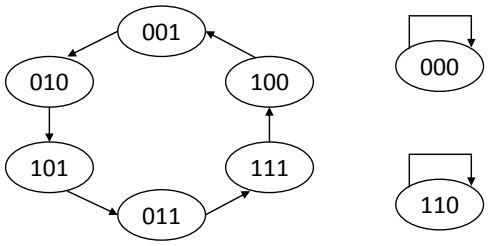

Fig. 1. The state transition graph of the mapping (2).

Each state transition function f induces a directed graph called the *state transition graph* of f in which the nodes represent the states and the edges represent possible transitions between the states. The node x (the predecessor) in connected to node y (the successor) by an edge if $f(x) = y$. Nodes forming a loop are called a *cycle*. A node x satisfying $f(x) = x$ is regarded as a cycle of length 1. Every node has a unique successor, but some nodes may have no predecessors and some may have more than one predecessor. In general, a state transition graph consists of a number of cycles and a number of directed trees rooted in cycles. For example, Figure 1 shows the state transition graph for the state transition function

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \rightarrow \begin{pmatrix} x_2 \\ x_3 \oplus x_1 x_2 \\ x_1 \oplus x_2 \end{pmatrix} \tag{2}$$

where "$\oplus$" is for the Boolean XOR (the Boolean AND in the product-terms is omitted).

If the state transition function f is invertible, so that for every y there is an x satisfying $f(x) = y$, then the state transition graph of f consists of pure cycles, without any trees rooted in them.

2.2 Stream ciphers

A stream cipher is a symmetric cryptographic primitive that allows two parties which share a secret key to communicate confidential information [42]. Stream ciphers generate a stream of pseudo-random bits, called *the keystream*, given a secret key and a public random initialization vector (IV). The keystream is combined with the plaintext message before the message is sent, typically by bitwise XORing. The received ciphertext is again XORed with the keystream to recover the original plaintext. As a result, the message cannot be read while in transit or storage by unauthorized parties who do not possess the secret key.

The stream ciphers considered in this paper are based on one or more shift registers with linear or non-linear state transition functions as well as an output function that maps the internal register state to keystream bits. Stream

ciphers have two working phases: (1) an initialization phase and (2) a keystream generation phase.

During the initialization, key and IV are typically mixed by shifting the shift registers while updating them with a combination of the state transition function and possibly also the output function (e.g. output function is used in Grain-80 and Grain-128, but not in Bivium and Trivium). During the keystream generation, the shift registers are shifted and their state transition functions are evaluated, while the keystream is generated from the internal state using the output function.

2.3 SAT solvers

SAT solvers are software tools that employ efficient heuristics to decide whether a set of constraints has a solution. Constraints are typically represented by a propositional formula in the Conjunctive Normal Form (CNF). In a CNF, each variable symbol in the expression, x or $\overline{x}$, is called a *literal*. A *clause* is a disjunction (Boolean OR) of literals. CNF is a conjunction (Boolean AND) of clauses.

SAT solvers are widely used in Electronic Design Automation (EDA) for formal verification, automatic test pattern generation, and logic synthesis [2], though they are also popular in a growing number of other domains.

Modern SAT solvers that are typically based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [43] which is a refinement of the earlier Davis-Putnam algorithm [44]. Most efficient DPLL-based SAT solvers include GRASP [45], where a powerful conflict analysis procedure was introduced, and Chaff [46], where a particularly efficient implementation of Boolean constraint propagation and a novel low overhead decision strategy were introduced. Solvers that employ these techniques are called *conflict-driven* SAT solvers. In this paper we use MiniSat [47], a conflict-driven SAT solver which is designed to be easily adaptable to different domains. Other reasons for choosing MiniSat is its competitive performance, code availability, and flexibility in adding arbitrary Boolean constraints

MiniSat employs a backtracking-based, depth-first search algorithm to find a satisfying assignment of variables for a set of clauses. The algorithm branches on a variable by picking it heuristically, guessing its value to be TRUE or FALSE and examining whether the values of other variables depend on this guess. The affected variables are then assigned and the search continues until no more assignments can be made. During this propagation phase, a conflict may be detected, i.e. a clause that cannot be satisfied may be found (because all its literals became false). If this happens, a so called *conflict clause* (or *learned clause*) which describes the wrong guesses that lead to the conflict is constructed and added to the SAT problem. Assumptions are then canceled by backtracking until the conflict clause becomes a unit, at which point it is propagated and the search process continues. The conflict clauses drive the backtracking, guiding the algorithm in choosing the best next guess, and speed up future conflicts by caching the reason for the conflict [47]. Eventually, either a satisfiable assignment is found, or the search tree is exhausted implying that no satisfying assignment exists.

3 Description of the Algorithm

The pseudocode of the presented algorithm is given as Algorithm 1. The algorithm 1 finds cycles of length k and factors of k induced by a state transition function $f : S \to S$ of a deterministic finite state machine with a set of states $S = \{0,1\}^n$.

3.1 Transition relation

The state transition function $f : S \to S$ can be described by a transition relation $T \subseteq S \times S$ which can be represented by the following propositional formula:

$$T(s, s^+) = \bigwedge_{i=1}^{n} (x_i^+ \ \leftrightarrow \ f_i(x_1, \ldots, x_n)) \tag{3}$$

where "$\wedge$" and "$\leftrightarrow$" are the Boolean logic operations AND and EQUALITY, respectively. Recall from Section 2 that $x_i \in \{0,1\}$ and $x_i^+ \in \{0,1\}$ are Boolean variables representing the bit number i of the current state $s = (x_1, x_2, \ldots, x_n)$ and the next state $s^+ = (x_1^+, x_2^+, \ldots, x_n^+)$, respectively, and $f_i : S \to \{0,1\}$ is the Boolean function computing the bit number i of the output of f, $\forall i \in \{1, 2, \ldots, n\}$.

For example, the transition relation for the state transition function in the example (2) is given by:

$$T(s, s^+) = (x_1^+ \leftrightarrow x_2) \wedge (x_2^+ \leftrightarrow x_3 \oplus x_1 x_2) \wedge (x_3^+ \leftrightarrow x_1 \oplus x_2). \tag{4}$$

3.2 Unfolding of transition relation

At step 2 of the algorithm, the unfolding of the transition relation T by k time steps is constructed.

The *unfolding* of a transition relation by k time steps is typically performed by taking the AND of the transition relations from the current state to the next state, from the next state to the one after next state, etc., from the state at the time step $k-1$ to the state at the time step k [30]. For instance, for the example defined by (2), the unfolding by two time steps is computed as $T(s, s^+) \wedge T(s^+, s^{++})$, where $T(s, s^+)$ is given by (4) and $T(s^+, s^{++})$ is given by

$$T(s^+, s^{++}) = (x_1^{++} \leftrightarrow x_2^+) \wedge (x_2^{++} \leftrightarrow x_3^+ \oplus x_1^+ x_2^+) \wedge (x_3^{++} \leftrightarrow x_1^+ \oplus x_2^+).$$

A possible satisfying assignment for the expression $T(s, s^+) \wedge T(s^+, s^{++})$ is $s = (101)$, $s^+ = (011)$, and $s^{++} = (111)$. This assignment corresponds to the path $101 \to 011 \to 111$ in the state transition graph in Figure 1.

Algorithm 1 An algorithm for finding cycles of length k and factors of k induced by a state transition function $f : \{0,1\}^n \to \{0,1\}^n$.

1: $T = \bigwedge_{i=1}^{n}(x_i^+ \leftrightarrow f_i(x_1, \ldots, x_n))$; /* transition relation induced by f */
2: $T_{-k\ldots0} = \bigwedge_{i=-k}^{-1} T(s_i, s_{i+1})$; /* unfolding of T from the time step $-k$ to 0 */
3: $F = T_{-k\ldots0}$; /* propositional formula representing $T_{0\ldots-k}$ */
4: $F = F \wedge (s_{-k} \leftrightarrow s_0)$; /* adding a constrain requiring the last state of a path */
5: /* of length k, s_0, to be equal to the first state, s_{-k} */
6: $C(s_0) = 0$; /* propositional formula representing the set states of all currently */
7: /* found cycles expressed in terms of variables of the state s_0 */
8: Find the set M_k of all factors of k;
9: **while** Sat(F) **do**
10: **for** each $m \in M_k$ **do**
11: **if** $s_{-m} = s_0$ **then**
12: **for** $(j = 0;\ j > -m;\ j--)$ **do**
13: $C(s_0) = C(s_0) \vee (s_0 \leftrightarrow a_j)$; /* marking a cycle of length m */
14: /*a_j is the assignment of variables of the state s_j returned by Sat(F)*/
15: **end for**
16: $F = F \wedge \neg C(s_0)$; /* adding a constrain describing marked cycles */
17: **break**;
18: **end if**
19: **end for**
20: **end while**
21: **return** all cycles contained in $C(s_0)$

3.3 Reverse direction of unfolding

Let $T_{0\ldots k}$ denotes the unfolding of the transition relation T from the time step 0 to the time step k:

$$T_{0\ldots k} = \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}).$$

Note that we need to switch from the previous notation $T(s, s^+)$ to the notation $T(s_i, s_{i+1})$ in order to describe longer paths.

It is common to unfold the transition relation T in the forward direction, as shown above, however, in our algorithm we have chosen to unfold T in the *reverse* order, from the time step $-k$ to the time step 0 (step 2):

$$T_{-k\ldots0} = \bigwedge_{i=-k}^{-1} T(s_i, s_{i+1}). \tag{5}$$

In this way, the states from the previous time steps rather than the next ones are added to the unfolding. As a result, the last state of the unfolded transition relation is always s_0, independently of the depth of unfolding. For example, the unfolding by two time steps is computed as $T_{-2,-1,0} = T(s_{-2}, s_{-1}) \wedge T(s_{-1}, s_0)$. In this case, the path $101 \to 011 \to 111$ in the state transition graph in Figure 1 corresponds to the satisfying assignment $s_{-2} = (101)$, $s_{-1} = (011)$, and $s_0 =$

(111). This way of unfolding reduces the number of extra clauses added at each iteration of SAT-solver to the propositional formula F representing $T_{0\ldots-k}$. A similar strategy was used in the algorithm [29].

3.4 Finding satisfying assignments

Once the transition relation is unfolded, the propositional formula F representing the unfolding $T_{-k\ldots0}$ is constructed using the standard procedure [30] (step 3). Then a constrain $s_{-k} \leftrightarrow s_0$ which means that the last state of a path of length k, s_0, should to be equal to the first state of the path, s_{-k}, is added to F (step 4) and the SAT-solver is called to find a satisfying assignment for F (step 9).

The function $\mathrm{Sat}(F)$ takes the expression F and returns TRUE if there exists an assignment of variables for the states $s_{-k}, \ldots, s_1, s_0$ which makes F true. Such a satisfying assignment corresponds to a path of length k in the state transition graph induced by the state transition function f. A path has length k if it makes k transitions between the states. Since we added to F the constrain $s_{-k} \leftrightarrow s_0$, we also know that in the path returned by the SAT-solver the last state is equal to the first. This means the we found a cycle which can either be of length k, or a factor of k.

To determine the length of the cycle, for each of the factors of k, m, we check if the last state of the path, s_0, is equal to s_{-m} (step 11). In the pseudocode, the set M_k containing all factors of k is computed at step 8, before the first run of the SAT-solver. If a shorter than k cycle if found, the **for**-loop breaks.

For instance, a possible satisfying assignment for the unfolding by two steps $T_{-2,-1,0} = T(s_{-2}, s_{-1}) \wedge T(s_{-1}, s_0)$ in the example (2) is $s_{-2} = (110)$, $s_{-1} = (110)$, and $s_0 = (110)$. This assignment corresponds to the path $110 \to 110 \to 110$ of length two in the state transition graph in Figure 1. We take the set of factors of two, $M_2 = \{1, 2\}$, and check its elements one-by-one to see if s_0 is equal to s_{-m}. After the first check, with $m = 1$, we can see that that this path corresponds to a cycle of length 1 and terminate the search.

3.5 Identifying and marking cycles

If Algorithm 1 finds that $s_0 = s_{-m}$ for some $m \in M_k$, the corresponding cycle of length m is marked by adding constraints to the input formula of the SAT-solver, F. In the following iterations, the algorithm only searches for paths in which the last state is not marked.

Clearly, if the last state of a path of length k does not belong to a cycle of length at most k, then no states of this path belongs to such a cycle either. In our case, the last state of a path is always s_0. Therefore, it is sufficient to express the constraints in terms of variables of the state s_0 only. By adding these constraints to F, we restrict F in such a way that no satisfying assignment for F returned by the SAT solver contains a state of a previously found cycle.

The constraints are added to F as $F = F \wedge \neg C(s_0)$ (step 13), where $C(s_0)$ is the propositional formula describing the characteristic function[3] of the set of states of all currently found cycles expressed in terms of variables of the state s_0 and "$\neg$" is the Boolean logic operation NOT. Before the first run of the SAT-solver, at step 6, $C(s_0)$ is initialized to 0. If the algorithm finds a cycle of length m (step 11), each of the m states of this cycle, s_j, $j \in \{0, 1, \ldots, m-1\}$, is added to $C(s_0)$ as $C(s_0) = C(s_0) \vee (s_0 \leftrightarrow a_j)$, where a_j is an n-bit vector corresponding to the assignment of variables of the state s_j returned by the SAT-solver.

In this way, the number of extra clauses added to F at each iteration is equal to the total number of states in the cycles found at this iteration, i.e. at most k. So, a formula of size $O(nk)$ is added to F, where n is the number of variables. Contrary, in a general case, a formula of size $O(np \log p))$, where p is the depth of unfolding, has to be added to to F to mark cycles [48].

If a satisfying assignment does not exist, then there is no cycle of length k or a factor of k among the unmarked states of the state transition graph. This implies that all cycles of length k and a factor of k are already found and marked. The algorithm terminates and returns the set of all identified cycles (step 21).

4 Analysis of Trivium, Bivium and Grain

We applied the presented algorithm to Trivium, Bivium, Grain-80 and Grain-128 stream ciphers. This section shows the results. We also compare the presented algorithm to the SAT-based algorithm from [29], which is the closest related work.

It might be worth mentioning that a random permutation over a set of p elements is expected to have an average of $\log p$ cycles [49]. The number of r-cycles is expected to decrease as $1/r$, for any fixed integer $1 \leq r \leq p$ [49].

4.1 Analysis of Trivium

Trivium [20] has a 288-bit internal state in which only 3 out of 288 bits are updated non-trivially. The rest of bits shift the content of the previous bit.

Let $(x_1, x_2, \ldots, x_{288})$ be the variables representing the bits of a current state of Trivium. At each clock cycle, the bit number i of the next state, denoted by x_i^+, is computed as:

$$x_1^+ = x_{288} \oplus x_{287}x_{286} \oplus x_{243} \oplus x_{69}$$
$$x_{94}^+ = x_{93} \oplus x_{92}x_{91} \oplus x_{171} \oplus x_{66}$$
$$x_{178}^+ = x_{177} \oplus x_{176}x_{175} \oplus x_{264} \oplus x_{162}$$

and $x_i^+ = x_{i-1}$ for all other x_i, $i \in \{2, 3, \ldots, 93, 95, \ldots, 177, 179, \ldots, 288\}$.

We searched for cycles of length k, for $k = 1, 2, 3, \ldots$ until the timeout of 12 hours was reached at $k = 31$. The results are shown in Table 1. The 2nd row in

[3] The *characteristic function* of a set is the function that takes the value 1 for elements in the set, and the value 0 for elements not in the set.

the table shows the number N_k of cycles of length k found by the Algorithm 1. The 3rd row shows CPU time, t. The experiments were run on a PC with Intel Core i7-4600U CPU at 2.1 GHz with 7.7 GB RAM running under Ubuntu 14.04 LTS.

k	1	2	3	4	5	6	7	8	9
N_k	1	0	21	0	0	0	0	0	0
t	0.004s	0.003s	0.011s	0.004s	0.014s	0.030s	0.040s	0.091s	0.099s

10	11	12	13	14	15	16	17	18
1	1	2	0	0	1	0	0	0
0.359s	0.261s	0.384s	0.034s	0.748s	0.565s	7.275s	10.604s	53.161s

19	20	21	22	23	24	25
0	0	0	0	0	0	0
1m2.358s	3m18.442s	3m5.661s	48.548s	0.285s	15m10.014s	14m43.513s

26	27	28	29	30
0	0	0	0	0
0.125s	8m46.997s	151m44.853s	0.239s	511m19.397s

Table 1. The number N_k of cycles of length k found in Trivium; t is the runtime.

In total, we found 27 cycles: one cycle of length 1 (all-0), 21 cycles of length 3, one cycle of length 10, one cycle of length 11, 2 cycles of length 12, and one cycle of length 15. We would like to point out that these cycles do not directly affect security of Trivium because none of them contains a state required for Trivium's initialization. Trivium is initialized by loading the following state into its state register:

$$(x_1, x_2, \ldots, x_{93}) = (K_1, \ldots, K_{80}, 0, \ldots, 0)$$
$$(x_{94}, x_{95}, \ldots, x_{177}) = (IV_1, \ldots, IV_{80}, 0, \ldots, 0)$$
$$(x_{178}, x_{179}, \ldots, x_{288}) = (0, \ldots, 0, 1, 1, 1)$$

where $(K_1, \ldots, K_{80})$ is an 80-bit key and $(IV_1, \ldots, IV_{80})$ is an 80-bit initialization value (IV), and clocking the cipher 4×288 times.

However, as we show in Section 5, short cycles can be used to mount a fault attack which results in a full secret key recovery.

4.2 Analysis of Bivium

We also applied the presented algorithm to Bivium stream cipher [50], a simplified version of Trivium.

Bivium [50] has a 177-bit internal state in which only 2 out of 177 bits are updated non-trivially. Using the same notation as in the previous section, Bivium

can be specified as:

$$x_1^+ = x_{177} \oplus x_{176}x_{175} \oplus x_{69} \oplus x_{162}$$
$$x_{94}^+ = x_{93} \oplus x_{92}x_{91} \oplus x_{171} \oplus x_{66}$$

and $x_i^+ = x_{i-1}$ for all other x_i, $i \in \{2, 3, \ldots, 93, 95, \ldots, 177\}$.

Due to a smaller search space, the timeout of 12 hours was reached at $k = 44$. In Bivium, we found only one cycle of length 1 (all-0) and 5 cycles of length 3. As in Trivium's case, these cycles do not directly affect security of Bivium because none of them contains a state required for Bivium's initialization. Bivium is initialized by loading the following state into the state register:

$$(x_1, x_2, \ldots, x_{93}) = (K_1, \ldots, K_{80}, 0, \ldots, 0)$$
$$(x_{94}, x_{95}, \ldots, x_{177}) = (IV_1, \ldots, IV_{80}, 0, \ldots, 0)$$

where $(K_1, \ldots, K_{80})$ is an 80-bit key and $(IV_1, \ldots, IV_{80})$ is an 80-bit IV, and clocking the cipher 4 times without producing keystream.

However, short cycles can be used to mount a fault attack which results in a full secret key recovery as we show in Section 5.

4.3 Analysis of Grain-80

The stream cipher Grain-80 [51] is constructed from an 80-bit NLFSR and an 80-bit LFSR. The LFSR is known to have the maximum period of $2^{80} - 1$ since it uses a primitive generator polynomial of degree 80. The period of the NLFSR is unknown, however, the input stage of the NLFSR is fed from the output stage of the LFSR to guarantee the lower bound of $2^{80} - 1$ on the overall period of the two combined shift registers.

We investigated the state space of the 80-bit NLFSR of Grain-80 for the case when it is disconnected from the LFSR. In this case, its state transition function is specified as

$$\begin{aligned} x_1^+ = {} & x_{17} \oplus x_{20} \oplus x_{28} \oplus x_{35} \oplus x_{43} \oplus x_{47} \oplus x_{52} \oplus x_{59} \oplus x_{65} \oplus x_{71} \oplus x_{80} \oplus x_{17}x_{20} \\ & \oplus x_{43}x_{47} \oplus x_{65}x_{71} \oplus x_{20}x_{28}x_{35} \oplus x_{47}x_{52}x_{59} \oplus x_{17}x_{35}x_{52}x_{71} \oplus x_{20}x_{28}x_{43}x_{47} \\ & \oplus x_{17}x_{20}x_{59}x_{65} \oplus x_{17}x_{20}x_{28}x_{35}x_{43} \oplus x_{47}x_{52}x_{59}x_{65}x_{71} \oplus x_{28}x_{35}x_{43}x_{47}x_{52}x_{59} \end{aligned}$$

and $x_i^+ = x_{i-1}$ for all other x_i, $i \in \{2, 3, \ldots, 80\}$.

The timeout of 12 hours was reached at $k = 46$. We found one cycle of length 1 (all-0), one cycle of length 3 and one cycle of length 12. Note that in the full version Grain-80, when the NLFSR is combined with the LFSR, the cycles of length 3 and 12 will translate into the cycles of length $3 \times (2^{80} - 1)$ and $12 \times (2^{80} - 1)$, given that the LFSR is initialized to any not all-0 state.

Grain-80 is initialized by loading an 80-bit key into the NLFSR, loading a 64-bit IV into the first 64-bits of the LFSR, filling the rest of LFSR with 1s, and clocking the cipher 160 times while updating the FSRs with a combination of the state transition function and the output function, without producing keystream. Since no restrictions are imposed on the 80-bit key loaded into the NLFSR, the key may potentially be one of the states of the short cycles.

4.4 Analysis of Grain-128

Similarly to Grain-80, the stream cipher Grain-128 is constructed from an 128-bit NLFSR and a 128-bit LFSR which is feeding the input stage of the NLFSR [21].

We investigated the state space of the 128-bit NLFSR of Grain-128 for the case when it is separated from the LFSR. In this case, its state transition function is specified as

$$x_1^+ = x_{128} \oplus x_{102} \oplus x_{72} \oplus x_{37} \oplus x_{32} \oplus x_{125}x_{61} \oplus x_{117}x_{115} \oplus x_{111}x_{110} \oplus x_{101}x_{69} \oplus x_{88}x_{80} \oplus x_{67}x_{63} \oplus x_{60}x_{44}$$

and $x_i^+ = x_{i-1}$ for all other x_i, $i \in \{2, 3, \ldots, 128\}$.

The timeout of 12 hours was reached at $k = 32$. We found one cycle of length 1 (all-0), one cycle of length 7 one cycle of length 8. Note that in the full version Grain-128, when the NLFSR is combined with the LFSR, the cycles of length 7 and 8 will translate into the cycles of length $7 \times (2^{128} - 1)$ and $8 \times (2^{128} - 1)$, given that the LFSR is initialized to any not all-0 state.

Grain-128 is initialized by loading an 128-bit key into the NLFSR, loading a 96-bit IV into the first 96-bits of the LFSR, filling the rest of LFSR with ones, and clocking the cipher 256 times while updating the FSRs with a combination of the state transition function and the output function, without producing keystream. Since no restrictions are imposed on the 128-bit key loaded into the NLFSR, the key may potentially be one of the states of the short cycles.

4.5 Comparison with the SAT-based algorithm from [29]

The closest related work to the presented algorithithm is the SAT-based algorithm from [29] which searches for all cycles in the state space, without restricting their length. It is designed for the analysis of $(n, 2)$ *random Boolean networks* (also called *Kauffman networks*) which are used in physics to model spin glasses [52] and in biology to study cell differentiation [53], immune response [54], and evolution [55]. The $(n, 2)$ random Boolean networks use non-invertible state transition functions $f : \{0, 1\}^n \to \{0, 1\}^n$ in which each Boolean function $f_i : \{0, 1\}^n \to \{0, 1\}$, $i \in \{1, 2, \ldots, n\}$, depends on at most two variables assigned at random. The $(n, 2)$ random Boolean networks are known to have a small number of short cycles (of the order of $\sqrt{n}$ for both, the number and the length [56]). While very successful for the $(n, 2)$ random Boolean network case, the algorithm from [29] is not suitable for the invertible mappings in which *all* 2^n states belong to some cycle. This makes the search for all cycles unfeasible for large n. Cryptographic algorithms typically use invertible mappings. For stream ciphers and hash functions this is important to prevent incremental reduction of the entropy of the state. For block ciphers, the round function has to be invertible in order to result in a unique decryption.

We have applied the algorithm from [29] to Trivium, Bivium, Grain-80 and Grain-128. In all four cases, it was able to find all-0 cycles, but no other short cycles. Shortly after the program start, the process was killed due to the memory blow-up.

5 Fault Attack Based on Short Cycles

The idea of fault attacks is to intentionally inject a fault in an implementation of a cryptographic algorithm so that the potential computational error caused by the fault can be exploited to expose secrets, see [35–37] for recent overviews.

In this section, we how short cycles of Trivium, Bivium, Grain-80 and Grain-128 can be used to mount a fault attack on their implementations which results in a full secret key recovery. The presented approach can be potentially applied to other cryptographic algorithms which are initialized similarly to these ciphers.

We assume that an attacker can inject transient faults in a selected flip-flop/gate. Moreover, we assume that the attacker can inject these faults at a precise time of his/her choice and more than once.

5.1 Trivium and Bivium

We start from Trivium and Bivium cases which are easier.

The attack is performed during the initialization phase as follows. Let $(C_1, C_2, \ldots, C_n)$, $C_i \in \{0, 1\}$, be values representing one of the states of a short cycle C of length m in the state space of the stream cipher under attack. Suppose that during the initialization phase the k-bit key $(K_1, K_2, \ldots, K_k)$ is loaded into the bits 1 to k of the internal state of the cipher. Then, rather than loading into the stream cipher the required initialization state, $n-1$ transient faults are injected so that the loaded state is $(K_1, C_2, \ldots, C_n)$ instead. The cipher is clocked as many times as required by the specification (1152 times for Trivium and 708 for Bivium), and then the keystream is generated and analyzed for the presence of length-m cycle. If there is a cycle of length m, then $K_1 = C_1$. Otherwise, K_1 is the complement of C_1. The attack is repeated $k-1$ times to recover the remaining bits of the key one by one. To recover the bit K_i, $n-1$ transient faults are injected so that the loaded state is $(C_1, \ldots, C_{i-1}, K_i, C_{i+1}, \ldots, C_n)$, for $i \in \{1, 2, \ldots, k\}$.

To summarize, an attacker can recover the complete k-bit key by performing k times an attack which injects $n-1$ transient faults.

The difficulty of implementing the presented fault attack depends on the mechanism which is used for loading the initialization state into a stream cipher. In the typical scenario when the bits of the initialization state are loaded sequentially, bit-by-bit, into the input stage of the register and then shifted further into the register, the attacker needs to identify in the layout of a chip the flip-flop corresponding to the input stage of the shift register, and, for each round i of the attack, $i \in \{1, 2, \ldots, k\}$, successively set this flip-flop to $n-1$ desired values C_j, $j \in \{1, \ldots, i-1, i+1, \ldots, n\}$, injecting one transient fault per clock cycle and keeping precise synchronization with the clock. Note that, if all-0 state is used as C, then the attack becomes particularly simple since the attacker has to inject the same value into the same position during $n-1$ clock cycles (mostly consecutive, with at most one interruption for K_i). As we mentioned in the introduction, feasibility of inducing transient faults at the same location in a successive manner by a laser has been demonstrated in [38].

Many stream ciphers, including Trivium, Bivium and Grain, can be parallelized to process more than one bits, m, at one clock cycle. In this case, the attacker has to simultaneously inject m faults into the m flip-flops corresponding to the first m stages of the shift register loaded in parallel. Depending on the type of fault injection mechanism used and the location of shift register stages in the layout, this can make the attack either simpler or more difficult.

In the case of parallel load, when the initialization state is loaded as one n-bit word in one clock cycle, the attacker has to simultaneously inject $n-1$ transient faults into $n-1$ different flip-flops.

5.2 Grain-80 and Grain-128

The case of Grain is more difficult because it updates its NLFSR and LFSR with a combination (XOR) of the state transition function and the output function during the initialization stage. If $K_i = C_i$ in $(C_1, \ldots, C_{i-1}, K_i, C_{i+1}, \ldots, C_n)$ and the value of the output function is 1, the NLFSR will not go to the next state of the state $(C_1, \ldots, C_{i-1}, K_i, C_{i+1}, \ldots, C_n)$ in the short cycle, but to its conjugate[4]. For an attack based on all-0 cycle, the value of the output function is 0, so the step we describe below is not necessary. For attacks using other short cycles, feedback connection from the output function to the XOR gate combining the state transition function and the output function has to be disabled. This can be done by injecting a fault which forces the value of the output function to logic-0. A fault injection attack using two lasers to inject two faults simultaneously at different locations has been reported recently in [39]. Therefore, an attack which injects one fault into the input stage of the shift register and simultaneously injects a second fault to disable the output function might be feasible. The fault disabling the feedback connection might be injected once for the duration of the whole initialization stage and all rounds of the attack, if this is easier to do.

Remind that we computed short cycles for Grain assuming that its NLFSR is disconnected from the LFSR. We can "disconnect" the LFSR by initializing it to all-0 state. In this case, the NLFSR state transition function will not be affected by the bit coming from the LFSR because the two values are combined by the XOR. Since feedback connection from the output function to the LFSR will be disabled during the initialization phase, the LFSR will never escape the 0-all cycle and therefore will not affect short cycles of the NLFSR.

To summarize, if $(C_1, C_2, \ldots, C_n)$ are values representing a state of a short cycle of the n-bit NLFSR of Grain and $(K_1, K_2, \ldots, K_n)$ is the key (for Grain it has the same length as the NLFSR), then, to recover the 1st bit of the key, we change the $2n$-bit initialization state to $(K_1, C_2, \ldots, C_n, 0, \ldots, 0)$ by injecting $2n-1$ transient faults in which $n-1$ have values corresponding to $C_2, \ldots, C_n$ and n have values logic-0. The rest of the attack is the same as for Trivium.

[4] Conjugate of a state $(C_1, C_2, \ldots, C_n)$ is the state $(\overline{C}_1, C_2, \ldots, C_n)$, where $\overline{C}_1$ is the complement of C_1 [57].

5.3 Countermeasures

The attack based on all-0 cycle can be detected using tests similar to the runtime health checks for monitoring true Random Number Generators (RNGs). FIPS-140 mandates that all approved RNGs use a runtime health check that verifies that the generator has not become stuck, i.e. generating the same value over and over again [58].

For the case when the bits of the initialization state are loaded sequentially, bit-by-bit, attacks using not-all-0 states short cycles can be detected by protecting the state using a parity check code. However, this countermeasure will not work if the stream cipher is parallelized to process an even number of bits, m, at one clock cycle and the attacker is injecting m faults in parallel.

6 Conclusion

In this paper, we presented a SAT-based algorithm for finding short cycles and applied it to Trivium, Bivium, Grain-80 and Grain-128 stream ciphers. We found that all four ciphers contain short cycles whose existence, to our best knowledge, was previously unknown. We described how short cycles can be used to mount a fault attack which recovers a secret key and discussed possible countermeasures.

Future work includes searching for short cycles in other cryptographic algorithms. Algorithms which use only a secret key and IV for their initialization without fixing some state bits to constants seem to be particularly vulnerable. We also plan to verify our assumptions by attacking FPGA implementations of Grain-128 and Trivium.

References

1. S. A. Cook, "The complexity of theorem-proving procedures," in *3rd Annual ACM Symposium on Theory of Computing*, pp. 151–158, 1971.
2. in *Logic Synthesis and Verification* (S. Hassoun and S. Tsutomu, eds.), Norwell, MA, USA: Kluwer Academic Publishers, 2002.
3. D. De, A. Kumarasubramanian, and R. Venkatesan, "Inversion attacks on secure hash functions using SAT solvers," in *Proc. of the 10th Int. Conference on Theory and Applications of Satisfiability Testing*, SAT'07, pp. 377–382, 2007.
4. V. Nossum, *SAT-based Preimage Attacks on SHA-1.* MSc. Thesis, University of Oslo, Norway, 2012.
5. P. Morawiecki and M. Srebrny, "A SAT-based preimage analysis of reduced Keccak hash functions," *Inf. Process. Lett.*, vol. 113.10-11, pp. 392–397, 2013.
6. I. Mironov and L. Zhang, "Applications of SAT solvers to cryptanalysis of hash functions," in *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, pp. 102–115, 2006.
7. C. Fiorini, E. Martinelli, and F. Massacci, "How to fake an RSA signature by encoding modular root finding as a SAT problem," *Discrete Appl. Mat.*
8. C. Patsakis, "RSA private key reconstruction from random bits using SAT solvers." Cryptology ePrint Archive, Report 2013/026, 2013. http://eprint.iacr.org/.

9. F. Lafitte, O. Markowitch, and D. Van Heule, "SAT based analysis of LTE stream cipher ZUC," in *Proc. of the 6th Int. Conference on Security of Information and Networks*, SIN '13, (New York, NY, USA), pp. 110–116, ACM, 2013.
10. F. Lafitte, J. Nakahara, and V. D. Heule, "Applications of SAT solvers in cryptanalysis: Finding weak keys and preimages," *Journal of Politics in Latin America;2014, Vol. 6 Issue 2, p1*, vol. 6, 2014.
11. B. Chen, "Strategies on algebraic attacks using SAT solvers," in *9th Int. Conference for Young Computer Scientists*, pp. 2204–2209, Nov 2008.
12. M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Proc. of the 12th Int. Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pp. 244–257, 2009.
13. P. Jovanovic and M. Kreuzer, "Algebraic attacks using SAT-solvers," *Groups Complexity Cryptology*, vol. 2.2, pp. 247–259, 2010.
14. N. R. Potlapally, A. Raghunathan, S. Ravi, N. K. Jha, and R. B. Lee, "Satisfiability-based framework for enabling side-channel attacks on cryptographic software," in *Proc. of Conf. on Design, Automation and Test in Europe*.
15. F. Massacci and L. Marraro, "Logical cryptanalysis as a SAT problem," *J. Autom. Reason.*, vol. 24, pp. 165–203, Feb. 2000.
16. D. Jovanović and P. Janičić, "Logical analysis of hash functions," in *Proceedings of the 5th International Conference on Frontiers of Combining Systems*, FroCoS'05, (Berlin, Heidelberg), pp. 200–215, Springer-Verlag, 2005.
17. T. Eibach, E. Pilz, and G. Völkel, *Attacking Bivium Using SAT Solvers*, pp. 63–76. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.
18. E. Homsirikamol, P. Morawiecki, M. Rogawski, and M. Srebrny, *Security Margin Evaluation of SHA-3 Contest Finalists through SAT-Based Attacks*, pp. 56–67. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.
19. L. Papay, *Use of SAT Solvers in Cryptanalysis*. MSc. Thesis, Comenius University, Bratislava, Slovakia, 2016.
20. C. D. Canniere and B. Preneel, "TRIVIUM specifications," citeseer.ist.psu.edu/734144.html.
21. M. Hell, T. Johansson, A. Maximov, and W. Meier, "A stream cipher proposal: Grain-128," in *IEEE Int. Symp. on Information Theory*, pp. 1614–1618, July 2006.
22. "eSTREAM: the ECRYPT stream cipher project," 2008. http://www.ecrypt.eu.org/stream/.
23. E. Dubrova, M. Teslenko, and A. Martinelli, "Kauffman networks: Analysis and applications," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 479–484, November 2005.
24. A. Garg, I. Xenarios, L. Mendoza, and G. De Micheli in *Proc. of 11th Int. Conf. on Research in Computational Molecular Biology (RECOMB'2007)*, Springer.
25. S. Bilke and F. Sjunnesson, "Stability of the Kauffman model," *Physical Review E*, vol. 65, p. 016129, 2001.
26. J. E. S. Socolar and S. A. Kauffman, "Scaling in ordered and critical random Boolean networks." http://arXiv.org/abs/cond-mat/0212306.
27. C. Oosawa, "Effects of alternative connectivity on behavior of randomly constructed Boolean networks," *Physica D: Nonlinear Phenomena*, vol. 170, no. 2, pp. 143–161, 2002.
28. L. Raeymaekers, "Dynamics of Boolean networks controlled by biologically meaningful functions," *Theoretical Biology*, vol. 218, no. 3, pp. 331–41, 2002.
29. E. Dubrova and M. Teslenko, "A SAT-based algorithm for finding attractors in synchronous Boolean networks," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 8, no. 5, pp. 1393–1399, 2011.

30. A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," *Proceedings of Design Automation Conference (DAC'99)*, pp. 317–320, June 1999.
31. T. Tamura and T. Akutsu, "An improved algorithm for detecting a singleton attractor in a Boolean network consisting of AND/OR nodes," in *Proceedings of the 3rd International Conference on Algebraic Biology (AB'08)*, vol. 5147 of *Lecture Notes in Computer Science*, pp. 216–229, Springer, July-August 2008.
32. T. Akutsu and T. Tamura, "On finding a fixed point in a Boolean network with maximum indegree 2," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E92.A, no. 8, pp. 1771–1778, 2009.
33. A. Naldi, D. Thieffry, and C. Chaouiya, "Decision diagrams for the representation and analysis of logical models of genetic networks," in *Proceedings of International Conference on Computational Methods in Systems Biology (CMSB'2007)*, vol. 4695 of *Lecture Notes in Computer Science*, pp. 233–247, Springer, September 2007.
34. S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, (London, UK), pp. 2–12, Springer-Verlag, 2003.
35. A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, pp. 3056–3076, Nov 2012.
36. D. Karaklajic, J. M. Schmidt, and I. Verbauwhede, "Hardware designer's guide to fault attacks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, pp. 2295–2306, Dec 2013.
37. R. Piscitelli, S. Bhasin, and F. Regazzoni, "Fault attacks, injection techniques and tools for simulation," in *2015 10th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pp. 1–6, April 2015.
38. E. Trichina and R. Korkikyan, "Multi fault laser attacks on protected CRT-RSA," in *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 75–86, 2010.
39. B. Selmke, J. Heyszl, and G. Sigl, "Attack on a DFA protected AES by simultaneous laser fault injections," in *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 36–46, 2016.
40. E. Dubrova, "A transformation from the Fibonacci to the Galois NLFSRs," *IEEE Transactions on Information Theory*, vol. 55, pp. 5263–5271, November 2009.
41. S. Golomb, *Shift Register Sequences*. Aegean Park Press, 1982.
42. M. Robshaw, "Stream ciphers," Tech. Rep. TR - 701, July 1994.
43. M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.
44. M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, pp. 201–215, July 1960.
45. J. P. M. Silva and K. A. Sakallah, "Conflict analysis in search algorithms for satisfiability," in *Tools with Artificial Intelligence, 1996., Proceedings Eighth IEEE International Conference on*, pp. 467–469, Nov 1996.
46. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, (New York, NY, USA), pp. 530–535, ACM, 2001.
47. N. Eén and N. Sörensson, *An Extensible SAT-solver*, pp. 502–518. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.
48. D. Kroening and O. Strichman, "Efficient computation of recurrence diameters," in *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'2003)*, vol. 2575 of *Lecture Notes in Computer Science*, pp. 298–309, Springer, January 2003.

49. P. Flajolet and A. M. Odlyzko, *Random Mapping Statistics*, pp. 329–354. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990.
50. H. Raddum, "Cryptanalytic results on TRIVIUM." ECRYPT Stream Cipher Project, Report 2006/039, 2006. http://www.ecrypt.eu.org/stream.
51. M. Hell, T. Johansson, and W. Meier, "Grain - a stream cipher for constrained environments," citeseer.ist.psu.edu/732342.html.
52. B. Derrida and H. Flyvbjerg, "Multivalley structure in Kauffman's model: Analogy with spin glass," *J. Phys. A: Math. Gen.*, vol. 19, p. L1103, 1986.
53. S. Huang and D. E. Ingber, "Shape-dependent control of cell growth, differentiation, and apoptosis: Switching between attractors in cell regulatory networks," *Experimental Cell Research*, vol. 261, pp. 91–103, 2000.
54. S. A. Kauffman and E. D. Weinberger, "The NK model of rugged fitness landscapes and its application to maturation of the immune response," *Theoretical Biology*, vol. 141, pp. 211–245, 1989.
55. S. A. Kauffman, *The Origins of Order: Self-Organization and Selection of Evolution*. Oxford: Oxford University Press, 1993.
56. M. Aldana, S. Coopersmith, and L. P. Kadanoff, "Boolean dynamics with random couplings." http://arXiv.org/ abs/adap-org/9305001.
57. H. M. Fredricksen, "Disjoint cycles from de Bruijn graph," Tech. Rep. 225, USCEE, 1968.
58. "FIPS PUB 140-2, security requirements for cryptographic modules," 2001. http://csrc.nist.gov/groups/STM/cmvp/standards.html.